View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Purdue E-Pubs

Purdue University

Purdue e-Pubs

Department of Computer Science Technical Reports

Department of Computer Science

1987

Mathematical Aspects of Scientific Software

John R. Rice
Purdue University, jrr@cs.purdue.edu

Report Number:
87-713

Rice, John R., "Mathematical Aspects of Scientific Software" (1987). *Department of Computer Science Technical Reports.* Paper 617.
https://docs.lib.purdue.edu/cstech/617

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.

MATHEMATICAL ASPECTS OF SCIENTIFIC SOFTWARE

John R. Rice

CSD-TR-713
September 1987

MATHEMATICAL ASPECTS OF SCIENTIFIC SOFTWARE

JOHN R. RICE*
Department of Computer Science
Purdue University
West Lafayette, Indiana 47907

CSD-TR-713

Abstract

The goal is to survey the impact of scientific software on mathematics. Three types of impacts are identified and several topics from each are discussed in some depth. First is the impact on the structure of mathematics through its role as the scientific tool for problem solving. Scientific software leads to new assessments of what algorithms are, how well they work, and what a solution really is. Second is the initiation of new mathematical endeavors. Numerical computation is already very widely known, we discuss the important future roles of symbolic and geometric computation. Finally, there are particular mathematical problems that arise from scientific software. Examples discussed include round-off errors and the validation of computations, mapping problems and algorithms into machines, and adaptive methods. There is considerable discussion of the shortcommings of mathematics in providing an adequate model for the scientific analysis of scientific software.

This is a preprint of a paper to appear in Mathematical Aspects of Scientific Software (J.R. Rice, ed.) Springer Verlag (1988), pp. 1-39.

MATHEMATICAL ASPECTS OF SCIENTIFIC SOFTWARE

JOHN R. RICE*
Department of Computer Science
Purdue University
West Lafayette, Indiana 47907

Abstract

The goal is to survey the impact of scientific software on mathematics. Three types of impacts are identified and several topics from each are discussed in some depth. First is the impact on the structure of mathematics through its role as the scientific tool for problem solving. Scientific software leads to new assessments of what algorithms are, how well they work, and what a solution really is. Second is the initiation of new mathematical endeavors. Numerical computation is already very widely known, we discuss the important future roles of symbolic and geometric computation. Finally, there are particular mathematical problems that arise from scientific software. Examples discussed include round-off errors and the validation of computations, mapping problems and algorithms into machines, and adaptive methods. There is considerable discussion of the shortcommings of mathematics in providing an adequate model for the scientific analysis of scientific software.

1. The Impact of Scientific Software on Mathematics

The goal of this paper is to survey the impact of scientific software on mathematics. Three areas are identified:

1) *The effect on the structure of mathematics*, on what mathematicians do and how they view their activities,
2) *New mathematical endeavors that arise*, new specialities or subspecialities of mathematics that may be created,

*This work supported in part by the Air Force Office of Scientific Research grant 84-0385.

3) *Mathematical problems that arise*, difficult mathematical problems or groups of problems arise from efforts to understand certain methods or phenomena of scientific software.

About 15 topics are presented which illustrate these impacts. No attempt has been made to be encyclopedic, the choices are those that appeal to the author. This introductory section is a summary of the survey, about a dozen of the topics are discussed in more depth in the later sections of this paper and only mentioned here. A few topics not discussed later are included here with a few remarks.

In considering the structure of mathematics, it is important to realize that not only does mathematics grow but that it also changes nature. Large subfields die out and not just because all the problems are solved or questions answered. There subfields become irrelevant to new directions that mathematics take. An example of this exists in scientific computation, namely making tables of mathematical functions. This endeavor started almost in the antiquity of mathematics and grew until there were a large number of practitioners. Volumes and volumes of tables were prepared and the methodology of creating, checking and making them easy to use became quite sophisticated. This endeavor is now in a steep decline because computers and scientific software have made it easier and more reliable to compute values from "first principles" than to look them up in tables.

Scientific software will lead mathematics to focus again more heavily on *problem solving and algorithms*. We need to analyze the intrinsic nature of problems, how hard they are to solve and the strengths of classes of algorithms. We need to examine again what it means to solve a problem, a step that will show many previous "solutions" to be of little value.

We need to examine again what it means to prove a result, and what techniques are reliable. Some groups in computer science have a substantially different view of proof then modern mathematical practice. They view proofs much more formally, somewhat reminiscent of Russell and Whitehead's Principia Mathematica. If computer programs are going to prove theorems, how does one prove the programs themselves are correct? For a delightful and insightful analysis of the pitfalls here, see [Davis, 1972]. I believe that we have learned several important things about proofs and scientific software. First, scientific software is not amenable to proofs as a whole because it contains many heuristics (i.e., algorithm fragments for which there is no underlying formal model, we just hope they work). Second, it is very difficult, often impossible, to say what scientific

software is supposed to do. Finally, the computing requirements for this approach are truly enormous.

We discuss the impact on mathematics of the creation and widespread use of *mathematical systems*. This development is now overdue and will have an enormous impact on education and practice of mathematics. It is plausible that one can automate large parts (the algorithmic parts) of mathematics from the middle elementary years to the middle undergraduate years. The results will be much more reliable abilities, less cost, more power for problem solving and more time to learn the mysteries of problem solving rather than the rote.

The largest and most visible new mathematical endeavor resulting from scientific software is that of *numerical computation*. It has a lot of structure and activity. The principle components are *numerical analysis*, a large subfield of mathematics and computer science, *mathematical software*, a smaller subfield of computer science, a part of *applied mathematics*, and *computational analysis*, a huge subfield of science and engineering. This endeavor has not yet had a large impact on most of mathematics. Perhaps this is because mathematics has turned away from problem solving and has been content to let other disciplines appropriate this endeavor. As the attention of mathematics is turned toward problem solving, interest in this area will rise substantially.

Sections 6 to 8 discuss the newer endeavors of *symbolic computation* and *geometric computation*. These are currently much smaller than numerical computation but may have a more immediate impact on mathematics because they are closer to currently active areas. Geometric computation is still quite immature and offers a host of challenges for mathematics.

Perhaps the best known problem area arising from scientific software is that of *round-off error analysis*. In 1948 John von Newmann and Herbert Goldstine pursued the idea of following the effect of individual errors and bounding the results. This is very tedious and overall this approach is a failure. A second and more subtle idea is to estimate the change in the problem data so the computed result is exact. This is often very practical and it gives very useful information, once one accepts that a realistic estimate of the error due to round-off is not going to be obtained. The third and most widely used idea is to do all computations with much higher precision than is thought to be required. One can even use exact arithmetic. This approach is quite, but not extremely, reliable. It is also expensive. The final approach has been to introduce *condition numbers*, these are essentially norms of the Frechet derivative of the solution with respect to the data. Note

that these do not take into account actual round-off errors but rather estimate the uncertainty of the solution in terms of the uncertainty of the problem data.

Round-off error analysis is somewhat unpopular because it is so frustrating and many people hope they can succeed in ignoring it. That is the real attraction of buying computers with long (e.g., 64 bit) word lengths, one gets high precision and, hopefully, freedom from worrying about round-off. Unfortunately there is a general lack of understanding of the nature of uncertainty effects in problem solving. There is confusion about the difference between the condition of a problem and that of an algorithm to solve it. If a problem is badly conditioned (the condition number is large) then nothing can be done about it while a badly conditioned algorithm might be replaced by a better one.

Condition numbers are sometimes misleading in that the estimates derived are grossly pessimistic. In my own work of solving elliptic PDEs, I see condition numbers like 10^5, 10^{10} or 10^{15} and yet observe almost no round-off effects. This leads to more confusion which is further compounded by the fact that scaling problems (simply changing the units of measurement) can have dramatic effects on round-off. This phenomena is poorly understood and difficult to analyze.

The problem of round-off error is just one aspect of a more general question: *How do you know the computed results are correct?* The mathematical results here are much less than satisfactory. Most problems addressed by scientific software are unsolvable within the framework of current mathematics. For example, given a program to compute integrals numerically, it is easy to construct a function (with as many derivatives as one wants) where the result is as inaccurate as one wants. Most theorems that apply to prove correctness of computed results have unverifiable hypotheses. And many theorems have hypotheses that are obviously violated in common applications. Most scientific software contains several heuristic code fragments. Indeed, it is usually not possible to give a precise mathematical statement of what the software is supposed to do.

The search for techniques to give better confidence in computed results is still on. A posteriori techniques still are not fully explored (it is easy to tell if x_o solves $f(x) = 0$). Computing multiple solutions efficiently is another technique that holds promise and which uses the old idea: Solve the problem 3 times (or k times) and with 3 methods and compare the results. The application of several techniques is usually required to achieve really high confidence in correctness. A rule of thumb is that it costs as much to verify the correctness of a computed result as to compute it in the first place.

Four other topics are discussed in Sections 9 to 12: 1) mapping problems and algorithms into the new parallel machines, 2) the analysis of adaptive algorithms, 3) how well mathematics models real problems, can one find theorems that are useful in assessing real computations, 4) the role mathematics plays in the experimental performance evaluation of scientific software. The final topic is particularly frustrating. Much like the weather, everyone talks about it but few do anything about it. One frequently hears statements "method x is the best way to solve problem y" which are in fact, little more than conjectures. Scientific and systematic performance evaluation is a lot of work, much of it is tedious and the work is not highly regarded by one's peers. No wonder that people prefer to do other things. We have the puzzling situation where research managers and funding agencies are always looking for "better" methods and yet they are uninterested in supporting work to measure which methods are actually good or bad.

2. Problem Solving and Algorithms

Historically, mathematics has arisen from the need to solve problems. It was recognized about a thousand years ago that one can codify the steps needed to solve some problems. These steps can be written down and someone can be told "*If you have this kind of problem, then follow these steps and you will have the solution.*" This idea matured into two of the most fundmental concepts inn mathematics: *models* and *algorithms*. Models arise from the need to make precise the phrase "*this kind of problem*" and algorithms make precise the phrase "*follow these steps*". Recall that, intuitively speaking, a model is an abstract system using axioms, assumptions and definitions which represents (well, one hopes) a real would system. An algorithm is a set of precise instructions to operate an abstract machine.

Mathematics has evolved through several well identified levels of problem solving. The lowest several of these are presented below along with typical problems and solutions.

Arithmetic

Problem	*Solution*
What is 2 + 2?	4
What is 7 × 8?	56
What is 1327/83?	15.9879518...
What is 3/2 × (1/8 − 3/5 + 1/12) / (37/8)?	47/310
What is $\sqrt{86}$?	9.273618...

Notes. There are many algorithms including memorization (table look-up in computer science terms) taught in school. It is significant that some studies suggest that about 80% of the entering college freshman cannot do long division, i.e., do not know an algorithm for computing 1327/83 as a decimal number. I would guess that a greater percentage of professional mathematicians and scientists cannot take square roots.

Algebra

Problem	*Solution*
What is $3x + 2y - x - 3y$?	$2x - y$
What is $(3x + 2y) \times (x + 3y)$?	$3x^2 + 11xy + 6y^2$
Solve $3x^2 - x - 7 = 0$	$x = 1.703...$
Solve $x^3 - 7x^2 + 3x - 110 = 0$	$x = 8.251...$

Note. Very few mathematicians know the algorithms for all of these problems.

Calculus

Problem	*Solution*
What is the derivative of e^x?	e^x
What is the integral of $\cot x$?	$\log \lvert \sin x \rvert + c$
What is the integral of $(\cos x) / x$?	$\log \lvert x \rvert + c - \sum_{i=1}^{\infty} \frac{(-1)^i x^{2i}}{2i(2i)!}$
What is the area under the curve $y = 1/(\sqrt{x}\,(1 + x))$ for x in $[0,\infty]$?	π
What is the series expansion of *erf* (x)	$\frac{2\pi}{\sqrt{x}} \sum_{i=0}^{\infty} \frac{(-1)^i x^{2i}}{(2i + 1)i!}$

Notes. The algorithms learned in calculus are rarely explicitly stated or taught. Problems are often called "solved" when one symbolic expression symbolic expression (say, a function or integral) is shown equal to another symbolic expression (say, an infinite series or product), neither of which can be computed exactly (even assuming one can do exact arithmetic with real numbers).

Beyond these three levels there are linear algebra, ordinary differential equations, combinatorics and others.

Even though the essense of mathematics is model construction and problem solving, a large percentage of the teaching effort is devoted to learning algorithms (often with neither the teacher or student being aware of this). I believe that it is much more important to *learn how to use knowledge* than to *learn knowledge*. In mathematical terms, it is more important to learn how to use algorithms than to memorize them.

The current situation is as follows:

(a) Enormous effort is invested in teaching people algorithms.

(b) People forget most of the algorithms they learn.

(c) Many algorithms of arithmetic, algebra, calculus, linear algebra, etc., can be implemented as scientific software and run on cheap machines.

(d) Many educators who expound the virtues of learning algorithms routinely use concepts, processes and models for which they do not know any relevent algorithms.

This situation is unstable and portends great changes in the educational system. This change will be slow but profound. Almost twenty years ago a compute program could make a grade of B on some calculus exams at MIT. Surely we cannot continue this when the cost of machines and software to do algorithms is becoming negligible.

It is fascinating to contemplate what the first two years of college mathematics would be if it were based on a mathematical system which includes the standard (and not so standard) algorithms of arithmetic, algebra, calculus, linear algebra, numerical analysis, geometry and combinatorics. The National Academy of Sciences is sponsoring a new study *Calculus for a New Century*, perhaps it will make some steps of change.

3. How Hard are Problems to Solve?

Since problem solving is one focus of mathematics, a central question is to determine just how hard various problems are to solve. Being hard to solve is measured by how much computation an algorithm must do, not by how hard it is to discover the algorithm. The problem of multiplying two integers, compute $a \times b$, illustrates the idea.

The "machine" to be used can multiply single digit integers and do addition of integers. The size of the problem is measured by the length N of a and b. The schoolboy algorithm requires N^2 single digit multiplications (every digit of a is multiplied by every digit of b) and N additions of long integers. This problem is thus possibly $O(N^2)$ hard, there might be no faster way to compute the product. Some thought leads to a method that only requires $O(N \log N)$ operations, it is much harder to show that it cannot be done with $O(N)$ operations (of standard abstract computing machines). So we know the intrinsic difficulty of computing $a \times b$ is not quite linear, $O(N)$, in the size of the problem and that the schoolboy algorithm is $O(N^2)$ but easy to remember.

There are some problems, often of a combinatorial nature, that are exponentially hard, i.e., the intrinsic work to solve these problems grows exponentially with the size of the problem. An example of such a problem is the *traveling salesman problem* where one is to compute the shortest route that visits each one of a set of N cities.

We give three practical questions in this area along with a few remarks.

1) *Are linear programming problems exponentially hard to solve?*

The simplex algorithm routinely solves these problems with N variables and constraints in roughly $O(N)$ steps. Yet an example has been discovered (after many years of search) where the simplex algorithm takes $O(2^N)$ steps. More recently, the discovery of algorithms for these problems whose worst case is only $O(N^7)$ or $O(N^3)$ or ?? has created somewhat of a sensation in the press. It is still not yet clear how hard these problems really are.

2) *Are there algorithms to solve partial differential equations (PDEs) that are as efficient as evaluating a closed form solution?*

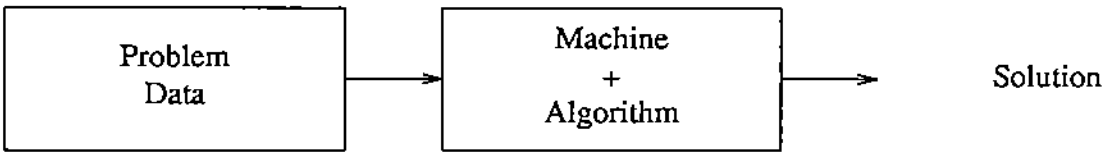
The theory of PDEs is one of the most sophisticated in mathematics and PDEs have long been regarded as among the most difficult problems to solve in applied mathematics. Dramatic progress [see, Rice, 1983, Chapter 10] has been made in developing better algorithms to solve PDEs and there is growing evidence that the answer to this question may be yes.

3) *Are systems of non-linear equations exponentially hard to solve?*

One sees examples of systems of thousands of equations being solved and yet one can construct rather simple, apparently well behaved systems with 10 variables that defeat all known algorithms. It may be that lots of large real-world problems are feasible to solve while the problem class as a whole is intractable. If so, it is of great interest to

discover why so many real-world problems are "easy" to solve.

There are two similar mathematical frameworks for studying this problem. The best known one is *complexity theory*. See, for example, [Aho, Hopcroft and Ullman, 1974]. A simple model of this theory is illustrated by the following schematic for problem solving:

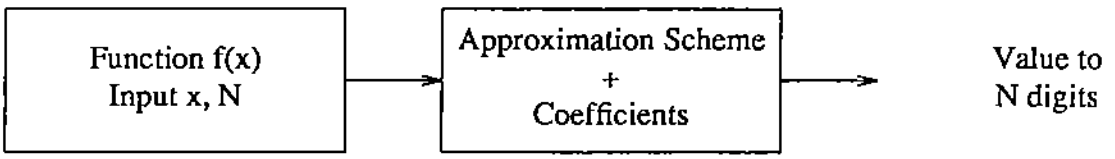


The central question here is: *for a given machine and problem class, find an algorithm which produces the solution with the fewest steps.* Simple examples of this theory are listed below (we are the notation that N is an integer, x is a real, f is a function, A is a matrix, and $\mathbf{x}$,$\mathbf{b}$ are vectors):

Problem	*Data*	*Machine*	*Solution Type*
Evaluate $f(N)$	N	Integer arithmetic (Turing machine)	integer
Multiply $a \times b$	a ,b N-digits	Addition 1 digit multiply	integer
Solve $A\mathbf{x} = \mathbf{b}$	matrix A vector $\mathbf{b}$ order N	Real arithmetic (Fortran)	vector of reals
Evaluate $f(x)$ to $N-digits$	real x $N-digits$	Real arithmetic	real

These four examples illustrate three facts about this theory: 1) The "power" of the machine must be carefully controlled (if the machine can multiply reals, then $a \times b$ is trivial). 2) The problem class must be carefully specified (if a and b are reals, then the problem of multiplying $a \times b$ is unsolvable with a machine that can only do single digit multiplication and addition). 3) It is nice to have one or two simple parameters that characterize the "size" of the problem (e.g., number of digits in input, order of matrix and vectors in input, number of digits in output).

The second framework is that of *approximation theory* (see, for example, [Feinnerman and Newman, 1974]) where the model of problem solving is



The central question here is: *For a given approximation scheme and class of functions f, find algorithms for the coefficients that give N digits of accuracy with the fewest coefficients*. This framework is less general than that of complexity theory but it is more general than it appears and has a much older and large body of results. Simple examples of this theory are listed below. We use the notation that C^k is the set of functions with k derivatives Lipshitz continuous on an appropriate domain, N is an integer, f is a function, x is a real, polynomials have degree P, L is a partial differential operator, Ω is a domain with boundary $\partial\Omega$.

Function Class (Problem)	*Data*	*Approximation Scheme (Machine)*	*Solution Type*
$f(x)$ continuous in [0,1]	N	polynomials	real
$f(x)$ in C^k	N	polynnomials	real
Analytic in unit disk	N	rationals of degree P/P	complex
Piecewise in C^3 + algebraic singularities	N	cubics with P pieces	real
$f(x)$ satisfies $L(f) = 0$ in Ω $f = 0$ on $\partial\Omega$	N	quintics with P pieces and in C^2	real

Observe that the approximation scheme defines the machines. Thus polynomials of degree P means the machine has the program

$V = c_P$
For $i = P - 1$ to 0 do $V = x \times V + c_i$
Value of $f(x)$ to N digits $= V$

where the c_i, $i = 0$ to N, are the coefficients computed by the algorithm of approximation theory. As before, one must be careful in formulating the problem properly so it will not be unsolvable or trivial. The approximation theory framework is more suitable for scientific software because it focuses on functions and reals rather than integers. The small subfield *analytic computational complexity* of computer science is a "merger" of these two frameworks. See, for example, [Traub, 1976] and [Traub and Wozniakowski, 1980].

As an aside, note that there is circularity in mathematics of the definitions of function classes and machines. Those functions which are C^k in [0,1] are exactly those functions where polynomials achieve accuracy of N digits with $10^{N/(k+1)}$ coefficients. Similar equivalencies exist for most other standard function classes of mathematics.

4. What is a Solution?

The classical solutions of problems are numbers or simple mathematical expressions. As problems have become more complex, we have admitted more complex solutions such as infinite series or products and integral forms. This raises the question of legitimacy for some such solutions are no solution at all. As a concrete example consider the Gamma and Incomplete Gamma functions. Consulting the *Handbook of Mathematical Functions*, National Bureau of Standards, Applied Mathematics Series, Vol. 55 one finds four integral representations, five asymptotic expansions, one continued fraction expansion and one infinite product. All of these are presented as solutions to the problem of evaluating $\Gamma(x)$. However, only the asymptotic expansions based on Stirlings Formula (coupled with the recurrence $\Gamma(x + 1) = x\Gamma(x)$) can be used to obtain actual values of $\Gamma(x)$.

These complex mathematical expressions are really just algorithms for solutions. And, as in the case of $\Gamma(x)$, they might work so slowly as to be useless. Nevertheless, they set the precedent for accepting an algorithm as the solution of a problem. Since algorithms are merely instructions for machines we can, in some cases at least, say that a machine is the solution of a problem. Thus, hand held calculators do solve the problems of arithmetic and elementary functions for "short" numbers.

Many problems are so complex that no exact solution will ever be found in the domain of mathematical expressions, algorithms or computer programs. Thus we must

settle for approximate solutions and formalize our thoughts about them. The series

$$e^x = 1 + x + x^2/2 + x^3/6 + \cdots + x^m/m! + \cdots$$

provides a sequence of approximate algorithms for the value of e^x which is quite satisfactory. The best one can hope for most scientific problems is scientific software of the same nature. The software has a parameter or two which, when increased like the m above, produce more accurate results with a reasonable added cost.

To illustrate the difficulties of defining what a solution is, consider the problem of solving partial differential equations (PDEs). We might find solutions which are

Finite: For example,

$$u(x,y) = 0 \text{ if } x < y$$
$$= (x-y)^3 \text{ if } x \geq y,$$

Closed Form: For example,

$$u(x,y) = e^x + y(\sin x - \cos y)$$

This almost never occurs so we might hope for solutions in

Symbolic (Analytic) Form: For example,

$$u(x,y) = \sum_{i=1}^{\infty} \sum_{j=1}^{\infty} \frac{i+j}{(ij)!} x^i \sin(jy)$$

$$u(x,y) = \sum_{i=1}^{\infty} \sum_{j=1}^{\infty} \frac{(1+\sqrt{i+j})(-1)^{i+j}}{(i+j)\log(i+2j)} x^i y^j$$

$$u(x,y) = \iint_R \sin^2(x-y) f(x,y) + \int_{\partial R} \cos(x)\sinh(y)\, g(x,y)$$

where $f(x,h)$ and $g(x,y)$ are known data functions. Such solutions might or might not

be useful. The first and third examples here probably lead directly to quite effective algorithms for $u(x,y)$, the second expansion is probably useless. The point is that these must be transformed to actual algorithms before one knows the "quality" of the solution.

A more desirable and realistic definition of a solution is to ask for a form as follows.

Algorithmic Form:

Read	Ndigits
Compute	K = EFFORT_ESTIMATE (Ndigits)
Compute	$u^e(x,y) =$ SOLVER (x,y,k)

where EFFORT_ESTIMATE and SOLVER are fixed procedures involving arithmetic and elementary functions (i.e., computer programs) and we know that

$$| u^e(x,y) - u(x,y) | \leq 10^{-Ndigits}$$

The infinite series symbolic solutions can usually be recast in this form with little effort.

More specific algorithms for PDEs are

Finite Difference Algorithm Form:

Read	Ndigits
Compute	Mesh = GRID_GENERATOR (Ndigits)
Compute	$u^e(i,j) =$ SOLVER $(i,j,Mesh)$

where

$$| u^e(i,j) - u(x_i,y_j) | \leq 10^{-Ndigits} \qquad x_i,y_j \text{ in Mesh}$$

Note that this does *not* solve the PDE as expected because a solution is provided only on a certain mesh of points. If that mesh only has two points in it, the solution is surely useless.

Finite Element Algorithm Form:

Read	Ndigits
Compute	Elements = ELEMENT_GENERATOR (Ndigits)
Compute	Coefficients = SOLVER (Elements)

then

$$u^e(x,y) = \sum_{i=1}^{k} (Coefficients\_i)\phi_i(x,y)$$

where the sum is over all k elements and the ϕ_i are basis functions such as cubic polynomial patches and where

$$| u^e(x,y) \_ u(x,y) | \leq 10^{-Ndigits}$$

Since finite element methods produce solutions defined everywhere, we do not have the difficulty seen for finite difference methods.

Pursuing this further in the case of PDEs, we see that an algorithm for solving a PDE should produce a second algorithm which evaluates the approximate solution. The desirable properties of the evaluation algorithm, call it UVAL, are as follows: Given (x,y) then $| UVAL(x,y) - u(x,y) | \leq 10^{-Ndigits}$ **and** UVAL is evaluated with constant effort (i.e., independent of x and y). This view allows a finite difference method to solve a PDE provided one is also provided with an interpolation procedure which preserves the accuracy of the table of approximate values. It may be difficult to find such an interpolation procedure.

We conclude from this discussion that, for many scientific problems, a mathematical analysis of the best ways to solve the problem requires a serious study of what a solution is.

5. What Should a Mathematical System Be?

At the conference there was a panel consisting of Carl de Boor, Bradley Lucier, Richard McGeehee and Clarence Lehman which addressed this question and generated lengthy discussion from the audience. It was generally agreed that is is practical today to build a computer system (both hardware and software) that

- automates the algorithmic aspects of undergraduate mathematics,
- communicates in standard mathematical notations and terminology,
- provides high resolution graphics support,
- costs much less than the equipment typically found in a scientist's lab.

While there was not agreement on the details of cost and performance, it is highly likely that for a few thousand dollars one could obtain high reliability in performing the algorithms, very substantial computing power for them and a much higher "thinking level" for computations and problem solving.

Most of the discussion involved opinions as to why such a system does not exist. It was agreed that it is obviously highly desirable to have such systems. There have serious research efforts on mathematical systems since the middle 1960's (see, [Klerer and Rheinfelds, 1968] for an early survey) and one might conclude that it is not yet known how to build them. This is not the case. One opinion is that the research groups spend most of their effort in adding sophisticated facilities rather than in building practical, usable systems. Several members of such groups indicated that they were, in fact, operating this way. This phenomenon has earlier been discussed by [Rice, 1973].

A more likely reason for the lack of such systems was advanced. Classify software into three groups as follows: 1) Basic tools that are essential to get anything at all done with reasonable effort. This includes language compilers, graphics packages, file systems, mail systems and so forth. 2) Problem solving tools that directly solve users' problems. This includes income tax packages, structural engineering systems, transactions systems for banks and so forth. 3) Generic tools that are widely applicable but usually not solvers of the final problems. This includes compiler generators, statistical libraries and mathematical systems. It is much harder to finance the generic software tools. Manufacturers of equipment must produce basic tools and end users are willing to pay good prices for problem solving tools but almost no one is willing to finance generic tools. This is quite reasonable in most cases, no one person or even group gets enough benefit themselves from these tools to afford the high development costs. A company that builds and markets generic tools finds that users will pay much less for them than for problem solving tools (which may be much cheaper to produce). Thus, no organization gets enough benefit from a mathematical system to justify its cost and software companies do not see users willing to pay enough to justify their risk in building and marketing it.

Other considerations were advanced and it is probable that lack of a good mathematical system is due to a complex combination of reasons. The difficulty is not with deciding what such a system should be but rather locating a mechanism to finance its development.

6. Symbolic Computing

Symbolic computing has established itself as a small, active subfield of computer science. However, it seems to me that it has not flourished as it should. A great deal of what mathematicians do is symbolic and symbolic computing has the potential for great impact on mathematics. Thus mathematicians should be concerned about why symbolic computing has not flourished. Before considering this, I make a few remarks about the field.

Symbolic computing has three major branches. The first branch is college algebra, calculus and applied mathematics. This is most relevant for scientific software as there are hordes who need to differentiate, integrate, expand in series, change coordinates, manipulate expressions, and so forth. The second branch is in abstract algebra where the operations of rings, fields, groups, etc., are involved. This branch is one that may eventually have the largest impact on mathematics. The third is in logic computations. In fact, I personally do not view this as part of symbolic computing. Its current inclusion is, I believe, due to people classifying everything non-numeric as symbolic. It is more reasonable to think of the subfields of computing as numeric (analysis), symbolic (algebra), logic, geometric (geometry/topology) and perhaps others.

I see four possible reasons for symbolic computing not to have matured as I expected. First is the lack of demand. There is less application of symbolic methods than numeric, but I do not think there is so much less. Great synergy can take place between numerical and symbolic computing and I think the time will come when people will not understand how they becomes so separated. Thus I do not consider the lack of demand as a primary reason.

Second it the lack of adequate computing power. This was a very significant problem in the 1960's and 1970's when memory was very expensive. Most computers simply had too little memeory for serious symbolic computing. This situation has changed dramatically as we now see ordinary computers with 8, 12, 24 or more megabytes of

memory, even simple workstations may have 4 megabytes, and personal computers with 1 or 2 megabytes of memory will be commonplace soon. So, while lack of computing power is no longer a barrier for the development of symbolic computing, it was for a 12 to 15 year period beginning in the early 1960's.

Third is the lack of appropriate languages or the incompatibility of symbolic processing with Fortran. It is true that Fortran must be substantially modified to allow for symbolic operations and it is plausible that some new language would be used instead of modifying Fortran. However, I believe that there is no magic in Lisp or even in functional languages in general. Serious symbolic computing has been done in completely Fortran environments and, more to the point, it is not so difficult to mix Lisp and Fortran in a disciplined way. This allows for symbolic computing to flourish in the midst of a numerical computing environment. Thus I do not consider language issues to be a primary reason.

Fourth is lact of appropriate computers systems. It is true that numerical computing has been Fortran based and symbolic computing has been Lisp based. And there have been computers built with special hardware to support Lisp. So it might be that the lack of Lisp machines has kept symbolic computing from flourishing. I do not consider the differences to be of primary significance and symbolic computing could have flourished without special hardware.

I conclude that there were unavoidable reasons for symbolic computing to develop slowly (primarily inadequate computer memories). Those reasons are now gone and mathematics can hope for rapid progress in this area.

7. Algorithms for Geometry

The algorithms of geometry are surprisingly complex. People look at pictures and do things (see patterns, intersections, move things) easily that are in fact very complex. Some ot these things cannot, as yet, be computed reliably. My message is simple: *the algorithms for geometry are much harder than one expects.*

The situation is further complicated because *there still is not a satisfactory way to represent general three dimensional objects for computation.* The objectives for representations are threefold:

- *Simplicity*: An object which is fairly simple should have a fairly simple representation.
- *Manipulation*: It should be easy (or at least feasible) to display objects, intersect objects, identify components (boundaries, holes, etc.) of objects or move them.
- *Generality*: One can accurately represent common objects and preserving important properties like smoothness or convexity.

We examine briefly the three principal representation techniques.

The *parametric representation* is two varieties: *explicit*, for example,

$$x = f_1(u,v,w), \quad y = f_2(u,v,w), \quad z = f_3(u,v,w)$$

$$u, v, w \text{ in unit cube}$$

and *boundary*, for example,

$$\text{side 1:} \qquad x = f_1(u,v), \quad y = f_2(u,v), \quad z = f_3(u,v)$$

$$u, v \text{ in unit square}$$

and the object is the interior of the region with boundaries side 1, side 2, side 3,etc. It can be a substantial effort to obtain a parametric representation (suppose the object already exists). The objectives of generality and simplicity are reasonably met but certain manipulations are computationally expensive. Examples of this include a) determining if a point is inside or outside, b) making intersections or contacts with other objects, and c) checking to see if the sides match properly to define an object.

The *functional* (algebraic or explicit) *representation* is to define the object as the set (x,y,z) so that $f(x,y,z) \geq 0$. It can be even harder to obtain a functional representation of an existing object than a parametric one. Some manipulations are easy (e.g., determining if a point is inside or outside) and there is no need to check the matching of sides. It is not clear how well the objectives of simplicity or generality are met.

The *constructive solid geometry representation* or building blocks is to take a small set of generic objects (e.g., spheres, cylinders, planes, parallel grids) and construct objects by unions, differences and intersections. For example (using – for difference and + for union) we might have

object = cube − (vertical cylinder) + spherical cap on east face

These representations are the easiest to manipulate and in many applications they are both simple and general. Their principle weakness is generality. It is difficult, for example, to represent a free form such as a face, a turbine blade or a car body smoothly, with modest accuracy and appropriate simplicity.

Since none of these representations is completely satisfactory, some applications use more than one which introduces the problems associated with transforming one representation to another. These transformations may be both computationally expensive and mathematically difficult to determine.

To illustrate the surprising complexity of algorithms for geometry, consider Figure 1 where a simple domain with four bounding sides is shown along with a rectangular grid. I have written a Fortran program to compute the points of intersection of the grid with the boundaries of the domain. This algorithm is very complex even though a person can identify the points very quickly. To quantify this, the code is over 2000 lines long which is more than enough to write many high performance, sophisticated PDE solvers on rectangular domains. Furthermore, the PDE solver will be more robust, this geometry code has been tested very extensively and yet cases of unsatisfactory performance still arise.

Not only are algorithms for geometry complicated, they are intrinsically hard. For example, consider the following two related problems:

1) Find a path for an object through an obstacle course.

2) Find the shortest such path.

We "solve" such problems everyday as we walk across campus, a dance floor or a parking lot. We feel these are easy tasks and even though we do not get the absolutely shortest path, we feel we come close. We are wrong, these are not easy tasks. Mathematical arguments of complexity show that such problems are intrinsically hard and experience at studying such problems confirms that there are no easy, quick ways to find shortest or almost shortest paths.

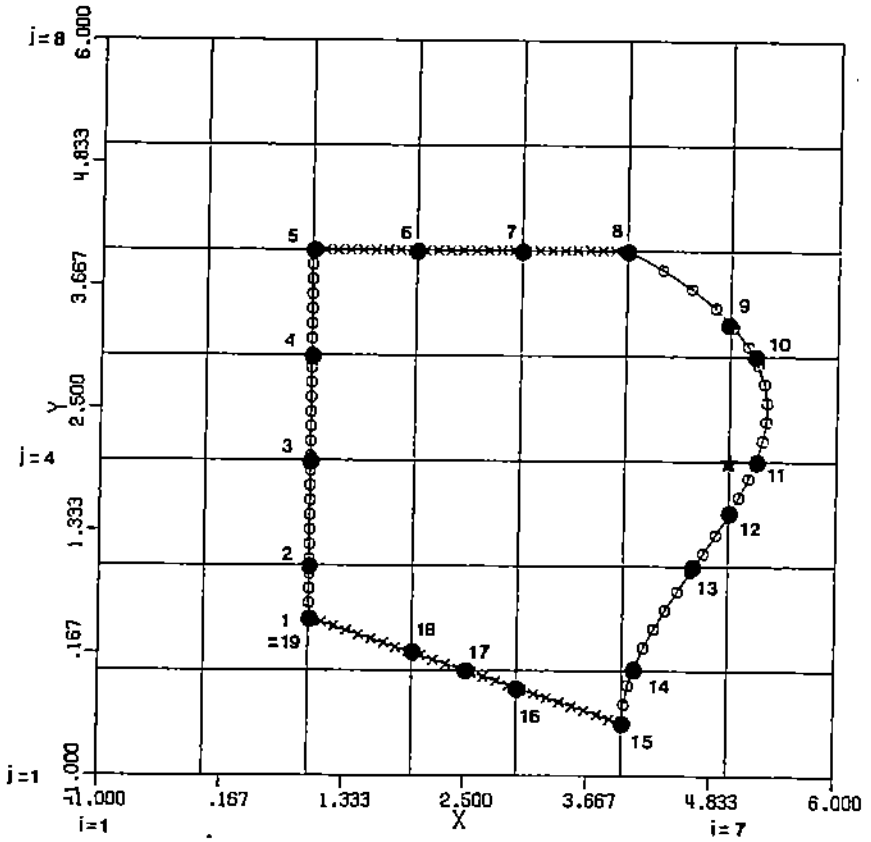


Figure 1. A domain showing the intersections of its four sides with a rectangular grid.

For another example, consider another pair of related problems:

1) Smooth off the sharp edges of an object nicely.

2) Blend 2 or 3 surfaces together smoothly where they join.

Again, in practice, people just "do" these things as they make or design objects. The Computer-Aided Design and Computer Aided Manufacturing (CAD/CAM) community has been challenged by this problem for about 30 years, ever since they started to automate some steps of design and manufacturing. Although algorithms have been found which work well in many applications, they are not robust and reliable. People are still trying to understand exactly what we mean when we say "smooth" or "blend" and how one can achieve this. The most recent advances on these problems have been by applying algebraic geometry techniques.

8. Geometric Computing

One of the exciting frontiers just opening up is geometric computing. The previous section discussed algorithms for geometry but these are things embedded in numeric or symbolic computing. We will see computations that directly manipulate geometric objects and geometric systems will include a wide variety of geometric and topological operators. One will not manipulate objects by applying operations (tediously detailed) to concrete representations but rather one will do manipulations explicitly in a geometric language.

There is no existing mature language for geometry beyond the colloquial one we use in everyday life. It is not at all obvious how the language will appear, but it must have:

Variables (objects):
lines, circles, boxes, spheres, ...
curves, surfaces, boundaries, tangents, corners, ...
regions, objects, pieces, interiors, ...

Attributes:
straight, smooth, convex, closed, ...

Operators:
rotate, move to, bend, join, add (union), ...
smooth, stretch, shrink, map, ...

I give an example geometric algorithm for a PDE problem.

Display domain A
Expand corner 3 to a side
Map results onto a rectangle
Do a boundary layer map toward side 2
Shrink domain locally toward point x
Overlay the domain with a rectangular grid
Transform the PDE on A to the current domain
Solve resulting PDE using spline collocation
Plot PDE solution on domain A

This algorithm is from real computations implemented in a numeric PDE oriented language. The code for that computation was much longer and less clear. To express this algorithm in a standard language like Fortran or Pascal would require thousands and thousands of lines of code.

Geometric computing requires even more computer power than symbolic computing. An adequate workstation needs perhaps 20 MIPS (million instructions per second), 5 MFLOPS (million floating point operations per second) of processing power (about 20 to 50 times a standard VAX 11/780 computer), color graphics displays with high resolution and sophisticated built-in graphics processing, 10-40 megabytes of main memory plus 1-5 gigabytes of auxiliary memories (disks). There will be millions of lines of Fortran or C or similar code (if such languages are used for the implementation). Hardware with these characteristics will cost perhaps $100,000 in 1988-89. Five to seven years later we can hope for such machines to cost 10 or 20 percent of this. The software will cost enormous sums and is unlikely to appear in 5 to 7 years.

Geometric techniques have always been one of the most powerful approaches to problem solving. There is a huge body of knowledge about geometry and topology but I doubt that is is well organized for the task of creating a system for geometric computing. Once this task gets under way, a host of problems will arise to challenge mathematicians and computer scientists. And, quantum leaps will be made in our power to solve problems.

9. Mapping Problems onto Machines

Our earlier discussion of problem solving centered on finding good or best algorithms for particular machines. These were rather simple abstract machines but, until recently, they modeled well the computers in actual use. The advent in the 1980's of complex machines using several – or hundreds of – processors in parallel has introduced a new and essentially difficulty into problem solving: *how do you map a problem's structure onto a machine to take advantage of the machine's power.* Worse that losing the simplicity of the old von Neumann architecture is that we must face dozens of different architectures, some radically different from others.

If we had enough time and money, we would take each pair of problem and machine and then analyze how to best create algorithms on this machine for this problem. This approach is used only for the most important problems and the related mathematical problems are similar to those of the slightly different approach presented next.

Realistically, we have only a few choices of reasonably efficient algorithms for a given problem. The practical approach is to take each of these algorithms and structure (or transform) them to a form that is quite flexible and amenable to mapping onto various machines. We call this a *computational structure* and it consists of:

- a precedence graph whose nodes are computational modules. These modules might range from a single or handful of instructions (as in microcoding of processors) to a collection of dozens of lengthy procedures (as in some scientific applications).
- information at each node of the graph on the processing time, memory and data access required by the modules,
- information at each arc joining nodes about the amount of data that must pass between the nodes.

A simple example of a computational structure is given in Figure 2. Figure 3 shows a realistic one with only part of the information given about a partial differential equation's (PDE) solver. See [Houstis, Houstis, Rice, 1987] for more details about this example.

One also has a *machine structure* which is a collection of processors and memories connected by a communication network. The traditional von Neumann machines have one processor, one memory and no communication network, the simplest possible machine structure. Figure 4 shows the schematic of a hypothetical, but currently plausible, machine structure. This machine has 31 ordinary processors (P) with 1 megabyte of local memory, 4 faster processors (P*) with 2 megabytes of memory, 4 vector processors (V), 14 modules with 4 megabytes global memory, 8 disks and 25 modules with 128 kilobytes of fast global memory. It also has a complicated communication structure.

The *mapping problem* may now be stated. Given computational and machine structures, determine an assignment of the computational nodes to processors, and memory and data nodes to memories so the whole computation is completed as fast as possible. It is well known that such scheduling or assignment problem are exponentially hard to solve, i.e., it is not feasible to find optimal solutions.

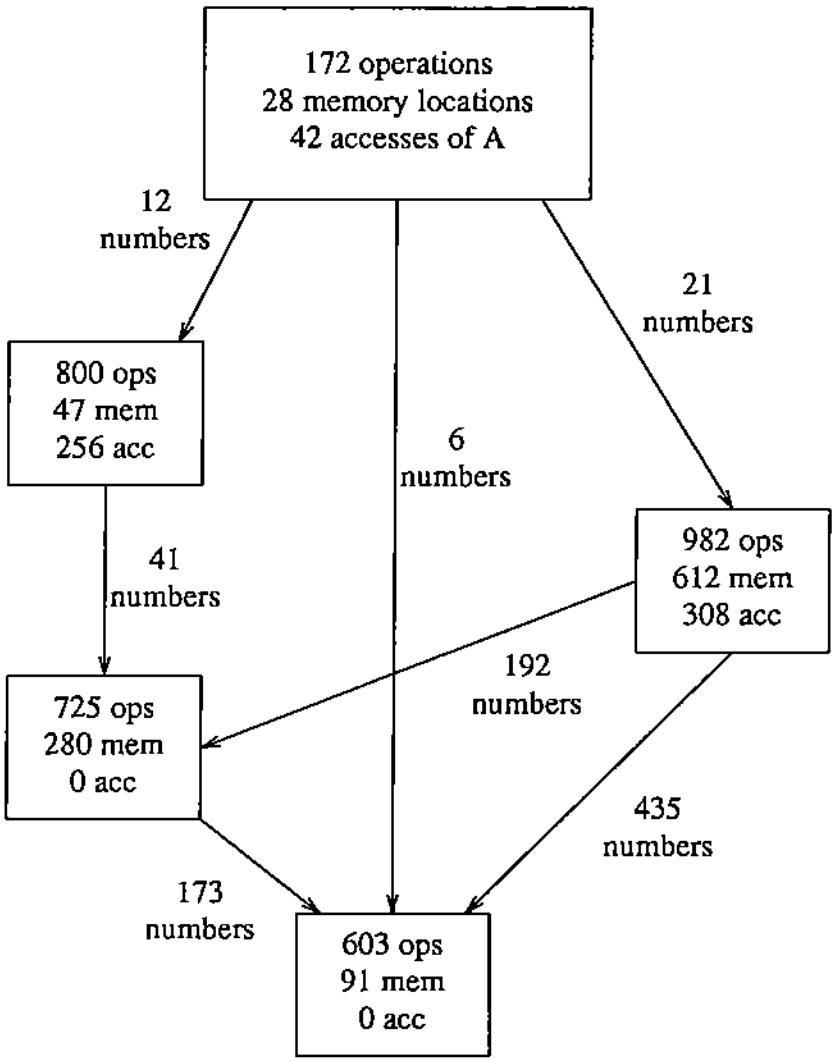


Figure 2. Simple example of a computational structure. Each box contains the number of operations (ops), memory locations (mem) and accesses made to the data labeled *A* (acc). Each arc is labeled with the data that passes between nodes.

There are two general approaches to solving this problem. First is to restrict the computational and/or machine structures to some simple class. Then optimal or nearly optimal mappings can be obtained by mathematical analysis. For example, one might assume that the computational structure is a regular tree and that the machine is a rectangular mesh of identical processors and memories. Second is to apply heuristic or approximate mapping algorithms. This can be done by the programmer while he is writing the program, by the compiler as the code is translated into machine language, by the loader as the pieces of the computational structure are organized to be placed in the machine, and during execution. In fact, sometimes one can apply all four of these

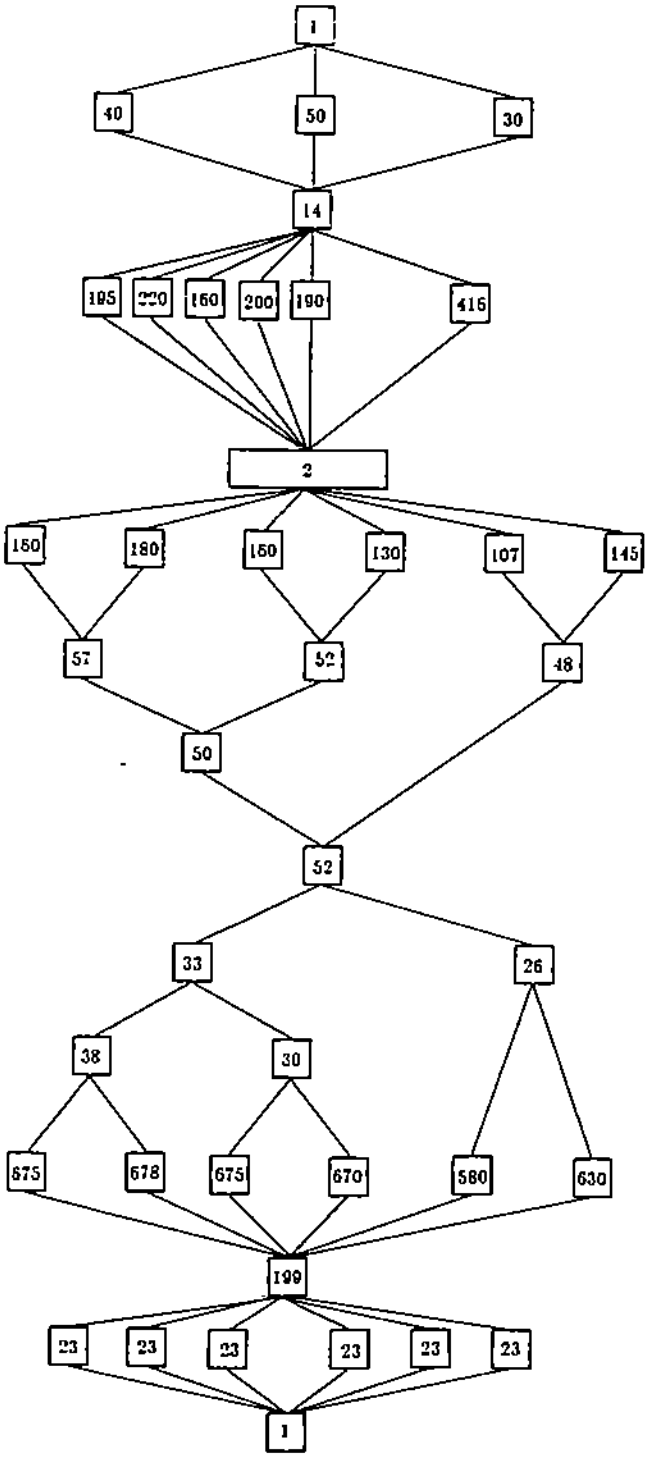


Figure 3. Partial computational structure of a PDE solver. The numbers at the nodes are the thousands of arithmetic operations to be performed there. The memory, data access and data passed between nodes is not shown.

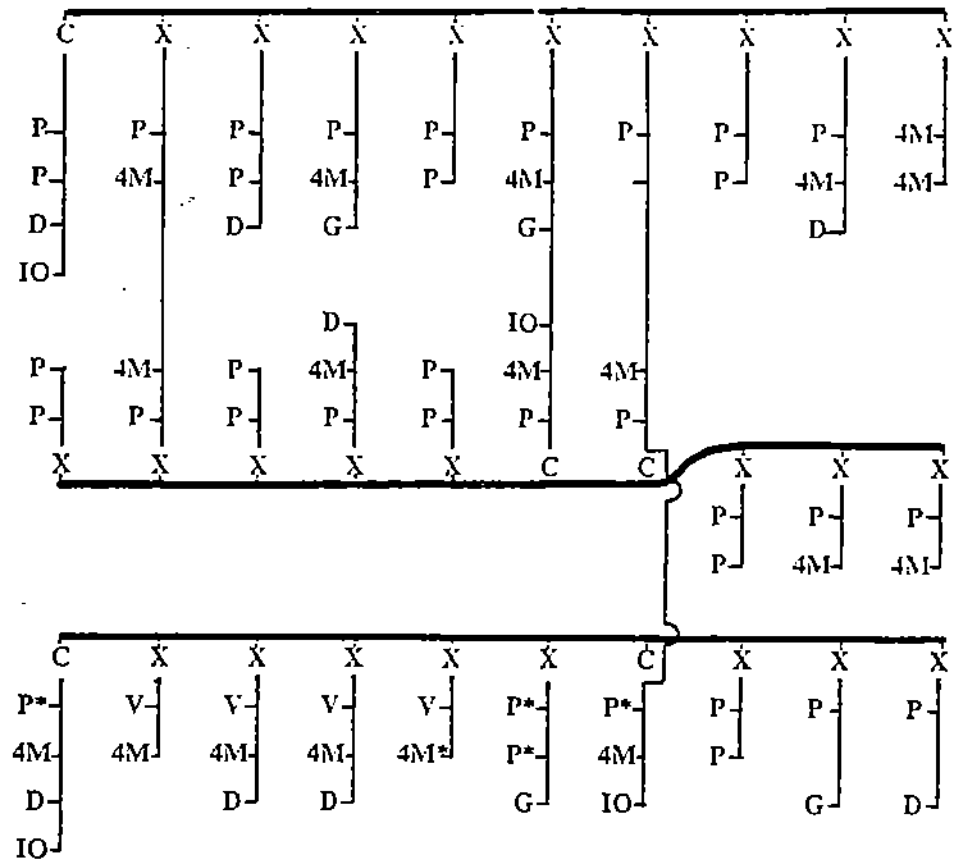


Figure 4. Schematic of a hypothetical machine with a complex architecture.

techniques to the same computation. Results from the first approach can be incorporated into the second to determine some pieces of the mapping. See the paper [Berman, 1988] in this volume for more discussion of the mapping problem.

Figure 5 shows the result of a heuristic method applied to the Cholesky factorization of symmetric matrices. The algorithm has been partitioned to be suitable for a machine with five processors. In Figure 5 the nodes are only numbered and the amount of data transferred between nodes is given. The machine has all identical processors and a simple bus communication network. The heavy lines show the nodes assigned to the processors.

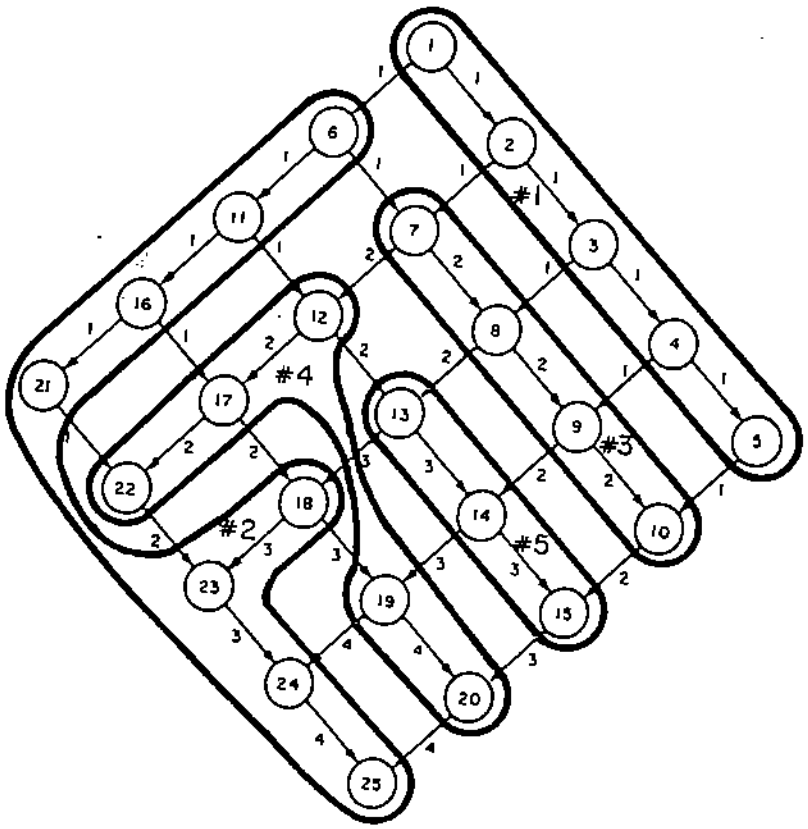


Figure 5. Computational structure of a parallel Cholesky factorization algorithm showing an assignment of the nodes to a machine with five processors.

10. Smart Algorithms or Adaptive Methods

The idea of having a numerical computation adapt itself to the problem at hand appeared in the early 1960's in a) ordinary differential equations programs where variable step sizes were chosen using local error estimators, and in b) numerical integration algorithms where intervals to be subdivided were chosen again using local error estimators. Before that computer programs used a fixed algorithm for all members of a problem class. Of course, when computing was done by hand (or desk calculator) adaptive methods were the norm. The human computer observed every step of the computation and adjusted many things to make them more efficient. These techniques were not unified into formal algorithms as required for computing machines.

Adaptive algorithms have led to dramatic advances both in theory and practice. For example, classical numerical integration of a function $f(x)$ with a singularity at x_o, say $f(x) = (\sqrt{x} + 3x^2)(\cos(x^2 - x^3)^2/(1 + 3\sqrt{x^2 + 1})$, requires one to locate the singularity,

determine its nature and use a special technique (e.g., Gauss quadrature with weight function, change of variable, series expansions, etc.). A modern adaptive integration algorithm can achieve good efficiency without knowing (as input) either the location or nature of the singularity. It determines both approximately during the computation and, as the accuracy requested increases, it achieves more accurate approximations. At the end of this section we sketch a simple, educational adaptive integration algorithm.

The idea of adaption has spread throughout numerical computation. For example, in solving partial differential equations one now sees time varying discretization meshs and different order discretizations being determined by adaptive criteria. Both these techniques come from the early 1970's algorithms for ordinary differential equations. These algorithms permitted, for the first time, the reliable and efficient solution of general initial value problems in ordinary differential equations. Similarly, these techniques are used in the amazingly robust and efficient algorithm for numerical integrations that were develop in the late 1970's and early 1980's. Of course, the application of these techniques in more than one dimension considerably complicates both the algorithms and their analysis.

Closely related is the concept of *polyalgorithm*, introduced in the middle 1960's in an attempt to achieve both efficiency and reliability. The idea here is to combine several basic algorithms along with various rules as when to switch from one basic algorithm to another. The rules are based on information gathered by the polyalogirhtm during its computations. For example, in solving $f(x) = 0$ one might start with the secant method which is quite efficient for most functions. But it does require $f(x)$ to be somewhat smooth and if the polyalgorithm senses that the secant method is failing to converge, it can switch to bisection (if two values of $f(x)$ are found with opposite signs) or simple systematic search for small $f(x)$ values. Polyalgorithms are widely used now, especially for optimization and nonlinear equations problems. They are also the precursors of *expert systems* for numerical computation. A big part of what an expert system should do is to choose or change the algorithm used and our experience with polyalgorithms shows that this is both feasible and productive.

These algorithms present mathematics with two challenges, one difficult and one revolutionary. The difficult challenge is simply to be able to analyze specific algorithms as applied to important problem classes. They are inherently more nonlinear and more complex than traditional algorithms and thus we should expect it to take a long time to develop suitable analysis techniques. However, progress is being made and one example

theorem is given in Section 12. That example is typical in that the hypotheses contain smoothness assumptions but these are not reflected in the conclusion. The reason for this is that the very foundation of the analysis has changed.

The revolutionary challenge is to replace the foundation of much of the function theory. Currently it is based on machines that can add, subtract, multiply and divide. That is, polynomials and rational functions are the models for functions. They are inadequate. In spite of the Weierstrass theorem, polynomials are unsuitable models for the functions that occur in the real world. Adaptive methods introduce logical decisions into the system or, in more analytic terms, piecewise rationals. There is overwhelming evidence that piecewise polynomials (or rationals) are very suitable models for real world functions. Once one switches to piecewise rationals (machines that add, subtract, multiply, divide and do logical tests) dramatic changes occur in the associated function theory. For example, the functions six(x), $(x-1)^{3.63}$, $\sqrt{x}$ and $(x-.15)^{1/69.23}$ are all in the same smoothness class. Here smoothness classes are defined (as in current mathematical practice) in terms of the approximation properties of the basic models, piecewise rationals.

Example: The adaptive trapezoidal rule

Consider a function $f(x)$ concave on [a,b] and the use of the trapezoidal rule to integrate it. Using five integration points (four intervals), we obtain the situation shown in Figure 6. The areas of the trapezoids below the curve are computed and added to estimate the area under the curve. The areas of the shaded triangles provide error bounds on the numerical integration. See [Rice, 1973] and [Rice, 1983, Section 7.5] for more details. A general framework for adaptive quadrature algorithm is given in [Rice, 1975].

The adaptive strategy now is to subdivide that interval with the largest triangle (the left one is thus subdivided next). This is continued until the error (sum of triangle areas) is as small as desired. This algorithm can be easily carried out with pencil, ruler and paper for a number of steps. One quickly sees that it adapts rapidly to the nature of $f(x)$.

We now test the effectiveness of adaption for the two functions shown in Figure 7. We compare this algorithm with the classical trapezoidal rule, Simpson's rule and a more sophisticated adaptive algorithm CADRE (it is also the program DCADRE used for

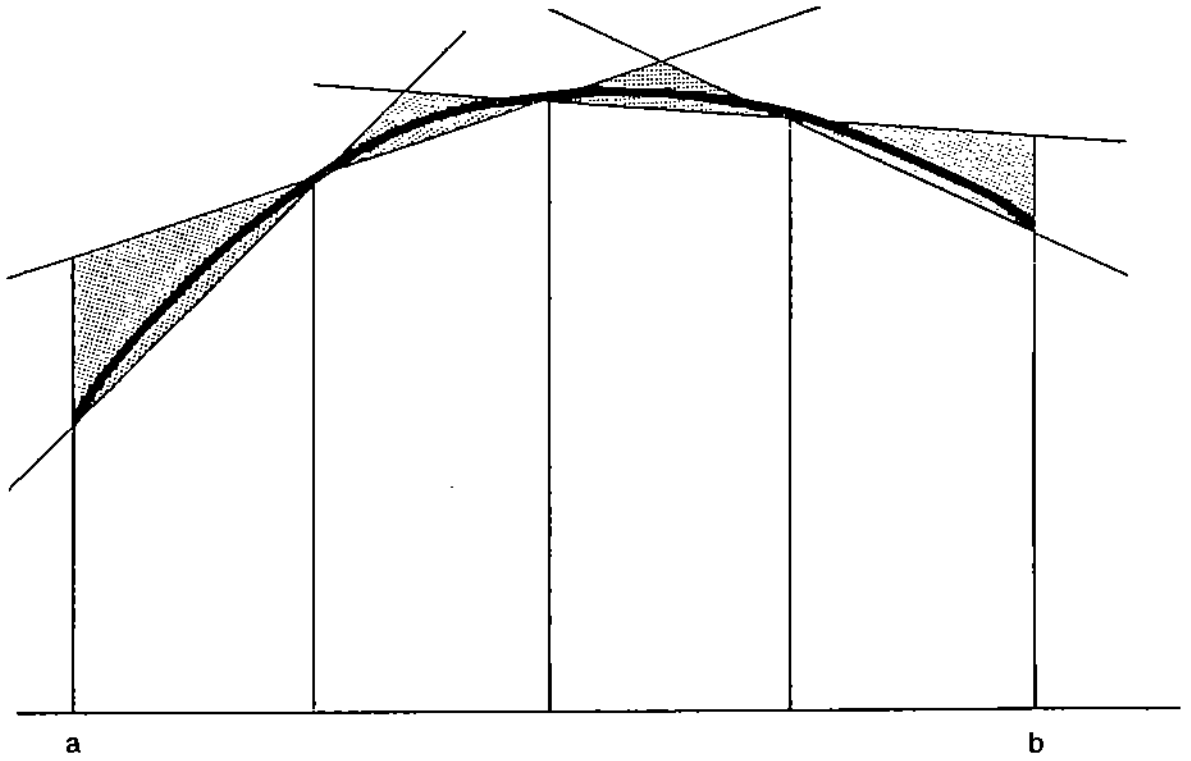


Figure 6. The geometry of the adaptive trapezoidal rule. The curve $y = f(x)$ lies in the shaded triangles whose areas provide error estimates.

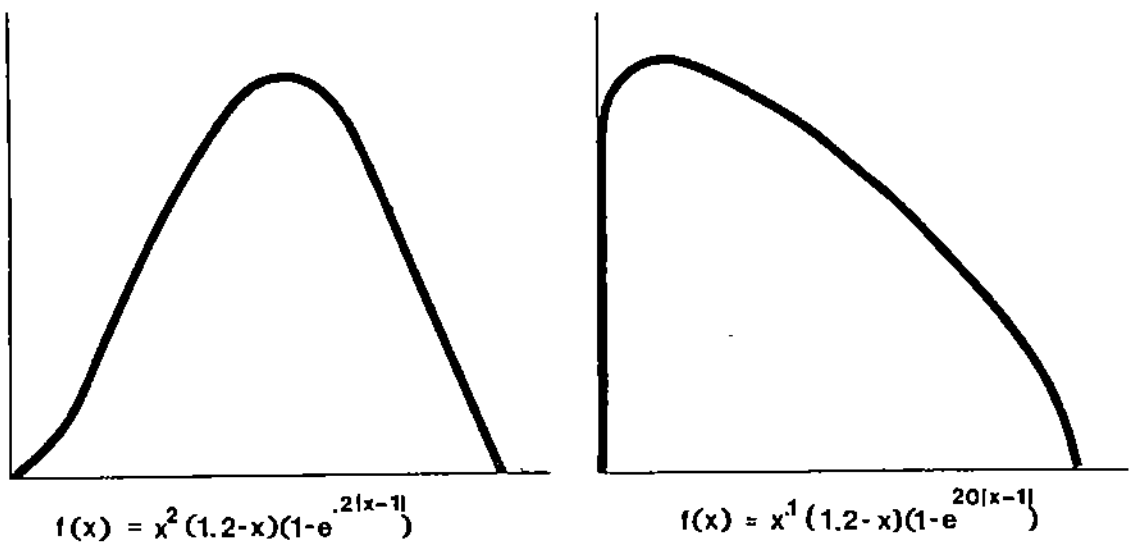


Figure 7. Two test functions for numerical integration, one (left) smooth and one (right) difficult to integrate.

many years in the IMSL library). We compare the algorithms' performance by counting the number of times $f(x)$ is evaluated in order to achieve a given level of accuracy. This ignores the difference in overhead among the algorithms which is not large for any of the algorithms.

Table 1. Results of numerical integration test. The number of $f(x)$ evaluations is given for each algorithm in order to achieve a given level of accuracy.

Smooth case, integral = .009549966

Method	**Accuracy** =	10^{-6}	10^{-7}	10^{-8}
Trapezoidal rule		68	180	580
Simpson's rule		13	26	66
Adaptive trapezoidal rule		22	64	208
CADRE*		33	33	41

Difficult case, integral = .602297

Method	**Accuracy** =	.01	.005	.001	.0005	.0001
Trapezoidal rule		36	67	285	510	2080
Simpson's rule		23	43	184	340	1470
Adaptive trapezoidal rule		14	19	43	60	133
CADRE*		33	33	33	33	33

*CADRE's accuracy is much better than the column heading

The test results in Table 1 show three things: 1) Adaptive algorithms can be very effective in handling difficult problems. 2) Adaptive algorithms perform better even for smooth problems, though not by a dramatic margin. 3) An adaptive algorithm might require more function evaluations for low accuracy results. This is because a good algorithm (such as CADRE) is rather cautious about accepting a result as correct; if we did not know the answer to these problems Simpson's rule would need more than 1470 evaluations to be confident of .001 accuracy for the difficult case.

11. Performance Evaluation: How Well Can We Solve Problems?

Since mathematics focuses on problem solving, it must address the question of how well we can solve problems. One issue is the performance of algorithms and this fits naturally into the framework of mathematical analysis. The earlier discussion of complexity theory, analytic complexity and approximation theory shows we have well developed machinery to analyze the performance of computations on abstract machines.

The situation is not nearly so satisfactory for computations on real machines. There are fundamental difficulties in each of the two area where mathematics might and should provide results. The first of these is *performance theorems*. Ideally we want a theorem of the following type: Consider problem class P defined to have properties $p_1, p_2, \cdots$. Then algorithm A always solves problem from class P.

In order for such theorems to be useful, it is essential that the classes P involved correspond well to real classes of problems. That is, mathematics must provide realistic models for real problems and it does not do this well. I estimate that half, at most, of real world problems can be reasonably modeled by currently standard mathematics. Further, a large portion of those problems modeled in applied mathematics do no correspond to any real problems. The difficulty comes primarily from the admissible classes of functions. In mathematics one has classes of functions defined by continuity, derivatives, analyticity, convexity, etc. In the real world one has classes of functions defined by smooth, well-behaved, ramp functions, the geometry of real objects, etc. Real world functions have a finite "scale", $f(x)$ might be smooth at a scale of 1, very rough at a scale of .0001 and randomly defined at a scale less than 10^{-6}. In mathematics, the "scale" of behavior (or classification) is zero except for properties like convexity. It is not clear whether all real world functions have six derivatives (or are entire) or whether none of them have. It is clear that there is a serious mismatch between models and reality.

A symptom of this difficulty is the problem of *unverifiable hypotheses* which is discussed in the next section. It suffices here to note that if one cannot say whether $f(x)$ has four derivatives or not then a theorem with this as a hypothesis is not useful.

It is unavoidable that performance evaluation must have a large component of experimental work. Our ability to analyze problems, algorithms and machines will always lag behind our ability to formulate problems, devise algorithms and build machines. Perhaps mathematics should not take on a role in experimental science, but it

(or perhaps statistics) should provide a proper framework for experimental studies.

One cannot overemphasize the importance of well conceived performance experiments. The traditional approach of consulting experts is unreliable. I believe the following is true (and I have observed concrete instances of this). Pose a simple problem P (e.g., solve 1 equation in 1 unknown in the interval [0, 1]), then consult 10 experts for the best method to use. About 4 to 6 methods will be recommended. If one consults the whole population of experts, almost the whole population of methods will be recommended. Science in general and scientific software is particular cannot advance without reiable information on the relative quality of its techniques.

An important contribution of performance evaluation is in the development of algorithms. In those areas where we have seen considerable advances in algorithms, it is extensive experimentation that has led to some of the key ideas. The paradym is that one creates an algorithm, evaluates its performance (speed or reliability) and then focuses on those instances where performance is poor. With some luck and perseverence, one discovers the "cause" of the poor performance and modifies the algorithms to improve it. The eventually best algorithms are discovered through systematic performance evaluation and were unknown beforehand.

Figures 8 and 9 show two simple examples of performance evaluation for algorithms to solve the Poisson problem on the unit square. In Figure 8, we have discretized the problem using standard, simple finite differences and then solved the resulting system of linear equations. We have plotted (for a particular problem) the accuracy achieved versus the computer time used for seven methods (see, [Rice and Boisvert, 1985] for specific definitions). We see that Gauss elimination (BAND GE) is the worst method, it is the algorithm that existed before 1945. Over the past 30 years we have discovered various iteration algorithms (represented by JACOBI CG here), fast Fourier transform algorithms (represented by FISHPACK-HELMHOLTZ and FFT-9 POINT) multigrid (represented by MULTIGRID-MG00) and others. The results of 30 years and perhaps 1000 papers is a speed up of about two orders of magnitude (a factor of 100) in solving this problem.

Figure 8 shows the results of using different discretizations for the same problem. This happens to be a rather easy problem, so high accuracy is obtained quickly compared to many realistic problems. The best result, multigrid, for ordinary finite differences is shown and we see that it is the worst algorithm. Speedups of five orders of magnitude are possible by using better discretizations. I conclude that the primary focus of research

effort should be on discretization rather than solving linear systems. This conclusion was reached by a number of people about 10 years ago and they began advocating it. Even today, now that the experimental and theoretical evidence is overwhelming, this conclusion is still only accepted by a small fraction of those solving partial differential equations. The reason for this is that the scientific software field has not matured enough to place scientific experimentations (performance evaluation) in its proper role.

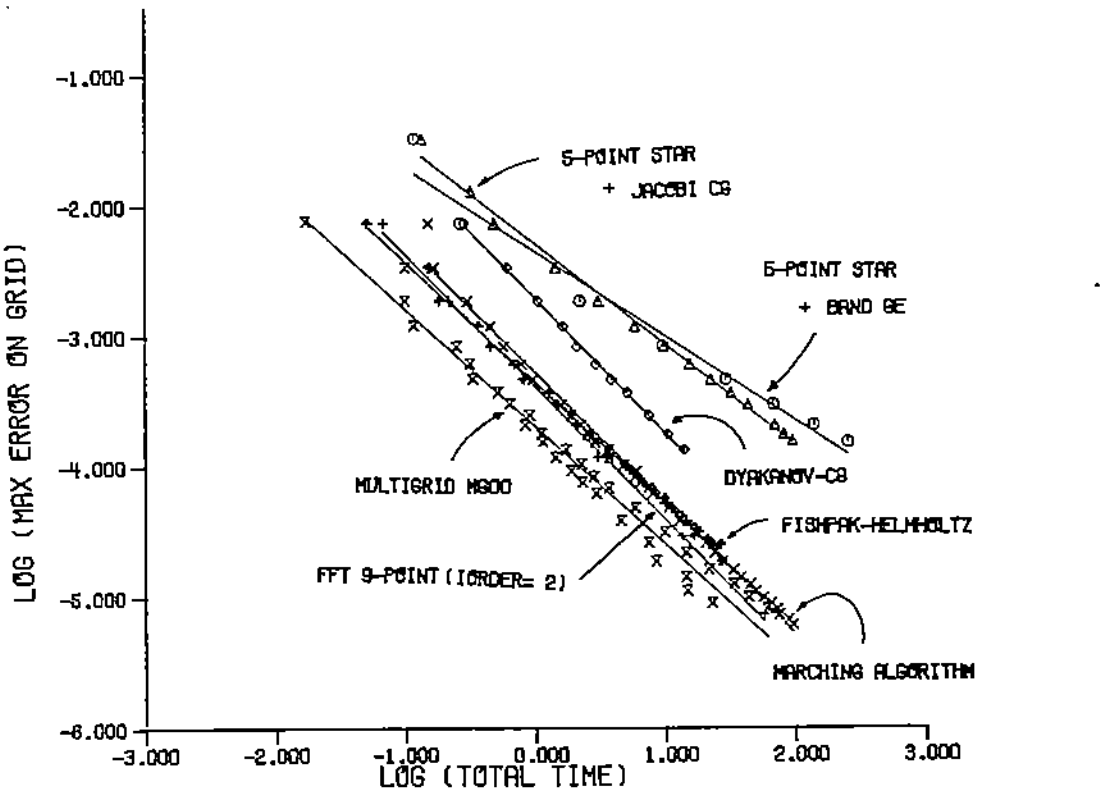


Figure 8. The performance (accuracy versus machine time in seconds) of seven methods to solve the linear system generated from discretizing $u_{xx} + u_{yy} = f(x,y)$ by standard finite differences.

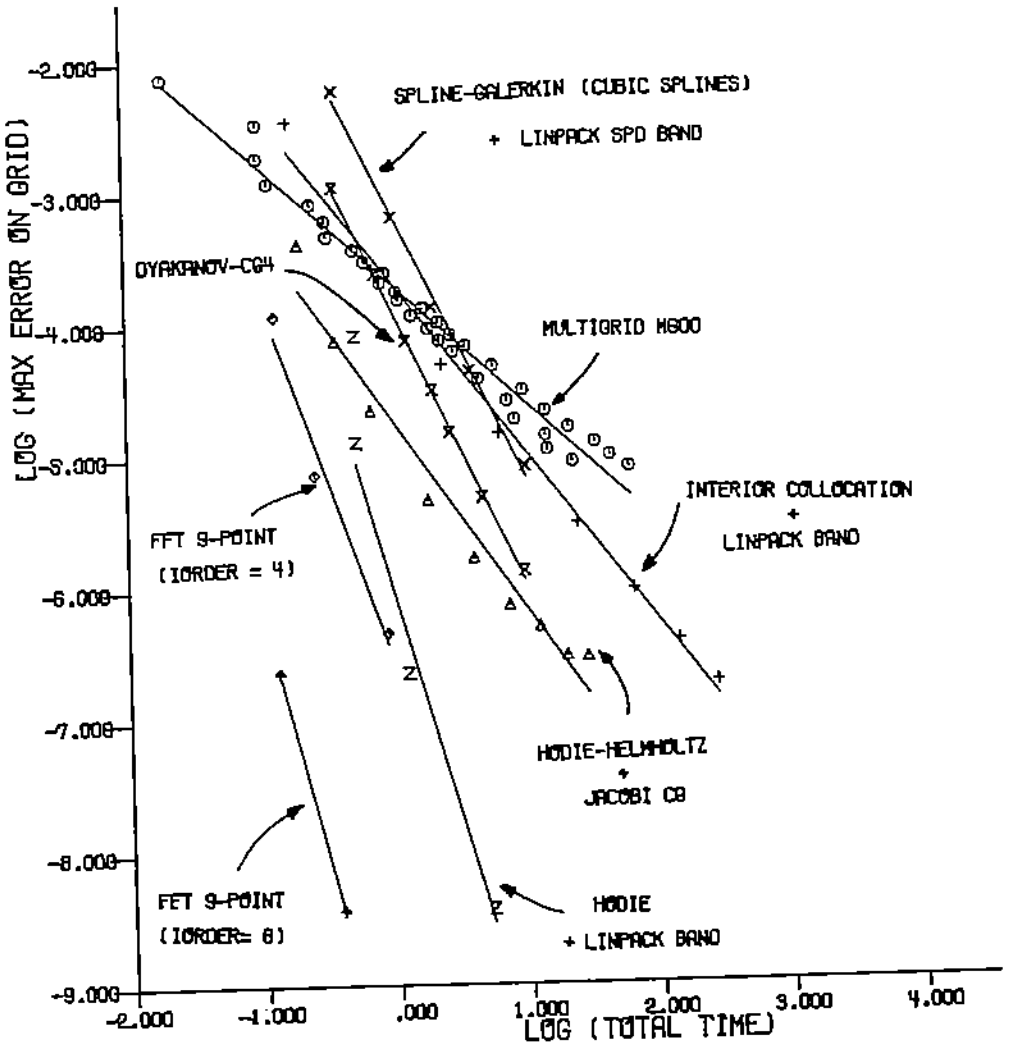


Figure 9. The performance (accuracy versus machine time in seconds) of eight discretizations plus linear equation algorithms to solve $u_{xx} + u_{yy} = f(x,y)$.

Once one accepts experimental performance evaluation as an essential part of scientific software, one faces another barrier. The approach is to state, for example:

Hypothesis: Algorithm A is more efficient than B is integrating functions on [0 , 1].

One then carries out an experiment by creating a population P of functions on [0 , 1] and performing the

Experiment: Randomly sample the population P, measure efficiency and draw conclusions, if any, from statistical test S.

The barrier arises at selecting the population P. If we select a mathematical population (e.g., functions with 3 continuous derivatives) then, with probability 1, members of this class are irrelevant to real world problems (e.g., their fourth derivatives have infinitely many discontinuities). This is just another symptom of the inadequacy of mathematics to model real functions. If we collect a thousand or a million functions from all kinds of real world applications, then we avoid some objections to the experiment but introduce others. For example, how do we know our collection process was not biased in some fundamental way?

We need statistical tests which apply to incompletely known populations. This is almost a contradiction in statistical terms, but this need is not restricted to the performance of scientific software. As a simpler example, assume we wished to show that DDT kills insects. It is estimated that only 1/3 of all insect species have been identified, so we have considerable difficulty here. Even if we showed that DDT kills all known species, the probability that DDT kills a given kind of insect might be only 1/3. Most people would find such a weak conclusion difficult to consider seriously.

12. Verifiable Hypotheses: Does Mathematics Model Reality?

The preceding two sections have raised the issue of whether traditional function theory provides an adequate model for real problem solving. The presentation suggested strongly that the traditional mathematical model is inadequate. Functions classified as "smooth" or "well behaved" by function theory may be unrealistic and pathological when viewed by a scientist while many functions he sees as "smooth" or "well behaved" are not so viewed by function theory. Concrete evidence of this mismatch occurs when one finds that a modestly accurate approximation (say good to 3-4 digits) to a simple curve (simple as seen by a scientist) requires a polynomial with degree in the hundreds or even thousands. Such polynomials are almost always intractable to compute or use and, perhaps worse, they have wildly oscillating derivatives where as the curve (or its underlying function) does not. Such curves occur in nature very commonly.

This issue appears again when we attempt to prove results about scientific algorithms. The typical theorem has hypotheses like

Let $f(x)$ have four continuous derivatives.....
Let $f(x)$ have $m + 1$ derivatives with.....
Assume $f'''(x)$ is bounded by k.....

Such hypotheses cannot be verified in practice and the resulting theorems may have little direct value.

There are directly conflicting plausible arguments about the "true" nature of real world functions. First, one can argue they are infinitely smooth except for a finite (small) number of discontinuies that correspond to discrete events like turning off a switch or changing composition of material. This argument leads directly to accepting piecewise rational functions as the appropriate mathematical models.

Second, we can argue that the real world is inherently discontinuous everywhere, its "microscopic" structure is either discrete or random or both. In any case, the mathematical definitions of continuity, derivation, etc., do not apply because, at some fine scale of examination, the functions are undefined or discrete or something intractible. The implication of this view is that the concepts of smoothness and behaviors of functions are related to a scale and that an adequate mathematical model must take this into account.

Third, one can argue that all real world functions are continuous and, hence, all their derivatives are also. This is supported by the proof that all *computable functions* are continuous, a standard result from theoretical computer science. The computable functions are, of course, the only functions that occur in scientific software and it is hard to accept, at least intuitively, that there are real world functions which are not computable. On the other hand, the function sign (x) looks very computable and discontinuous. This theorem stands in direct contradiction to the first intuitive view and it makes much of modern mathematics irrelevant. This fact underscores that we do not yet understand the true nature of the relation between computation, mathematics and the real world.

I cannot resolve these contradictions but the fact that mathematical function classes fail to adequately model reality is leading to more and more difficulty as we attempt to make better analyses of problem solving and scientific computations. As an example, I point out that a number of theorems have been published which "prove" that adaptive computation methods work no better than non-adaptive ones. Such nonsense stems directly from the shortcommings of mathematics in modeling real world problem solving.

As an example of a small step toward verifiable hypotheses, I state a theorem on the convergence of the adaptive trapezoidal rule briefly presented in the preceding section.

Let

$$\text{If} = \int_a^b f(x)dx$$

$$Q_N f = \text{Quadrature result using } N \text{ values of } f(x)$$

Assume that

a) $f(x)$ has p continuous derivatives except for a finite number of algebraic singularities $s_i, i = 1$ to k. Further

$$|f^{(p)}(x)| \leq K \prod_{i=1}^{k} (x - s_i)^{\alpha - p}$$

holds for some constant K and $\alpha > -1$.

b) we know a characteristic length (scale) $\lambda(f)$ so that local error estimates are valid on intervals of length less than $\lambda(f)$.

Theorem [Rice, 1975]. There are adaptive quadrature algorithms so that, for some constant C,

$$|\text{If} - Q_N f| \leq C/N^p$$

Corollary: Let $P = 2$ and $\lambda(f) =$ one fifth the minimum distance between inflection points, cusps, the s_i, a and b. Then the theorem holds for the adaptive trapezoidal rule algorithm.

The unusual feature of this theorem is that it makes a direct scale assumption. If one views all functions as piecewise smooth (as many do), then the hypotheses make explicit that one must identify all the discontinuities and then one can proceed with confidence to a computation. The scale $\lambda(f)$ is used by the algorithm and a more satisfactory result would show explicitly how the constant C depends on $\lambda(f)$ and p.

REFERENCES

Aho, A.V., J.E. Hopcroft and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison Wesley (1974).

Berman, F., The mapping problem in parallel computation, this volume, (1988).

Davis, P.J., Fidelity in mathematical discourse: Is one plus one really two?, *Amer. Math. Monthly*, 79(1972), 252-263.

Feinerman, R.P. and D.T. Newman, *Polynomial Approximation*, Wilkins and Wilkins (1974).

Houstis, C.E., E.N. Houstis and J.R. Rice, Partitioning PDE computations: Methods and performance evaluation, *J. Parallel Computing*, **5** (1987), 141-163.

Klerer, M. and J. Reinfelds, *Interactive Systems for Experimental Applied Mathematics*, Academic Press (1968).

Rice, J.R., NAPSS-like systems: problems and prospects, *Proc. Nat. Computer Conf.* (1973), 43-47.

Rice, J.R., An educational adaptive quadrature algorithm, *SIGNUM Newsletter*, **8** (April 1973)

Rice, J.R., A metalgorithm for adaptive quadrature, *J. Assoc. Comp. Mach.*, **22** (1975), 61-82.

Rice, J.R., *Numerical Methods, Software and Analysis*, McGraw Hill (1983).

Traub, J.F. and H. Wozniakowski, *A General Theory of Optimal Algorithms*, Academic Press (1980).

Traub, J.F., *Analytic Computational Complexity*, Academic Press (1976).